Keymap.h - Evaluation

Student #20107104674840

Write a brief (1-2 pages) evaluation report of your map class: justify any design decisions (e.g. its interface, the use of friends) and evaluate the support provided by C++ for developing abstract data types and the use of this support for your map class.
Identify any weaknesses in your implementation and discuss improvements and alternative representations, and compare your map class with the map class provided in the STL

Design Decisions

The specific implementation of Keymap produced in this assignment utilises two parallel arrays of pointers in order to store data; that is to say, the map doesn't directly store data, it simply stores the locations to the data. This should make manipulations of the map, updating, removing, etc, relatively efficient due to the small size of memory pointers, when compared to the data they point toward. This decision was made to reduce duplication of objects in memory, in theory leaving a smaller footprint.

Similarly, some keymaps wrap the `key:value` pair, and then store the wrapped pair in a single array. I opted against this

There are four different `insertPair` methods in my implementation, and the aim was to cover as many

Insertion is linear and removal preserves order; insertion as a result is fast, as it simply checks the current `length` of the array, and inserts the pair at `array[length+1]`. Removal requires finding the element, and copying everything from the right-hand-side of the element one location to the left, which can be especially slow on larger datasets.

The map is iterable both forward and backward, and

As the C++ keyword `NULL` compiles to `0`, I opted to not dissallow "null" values, however the program will not accept insertion of `nullptr`s into the map.

Improvements

While the insertion is fairly lightweight, my delete method could be improved significantly. The current implementation iterates over the entire array twice, and copies elements to a new array while skipping over the element to delete. This operates in the order of `O(2n)`, which works fine for smaller datasets however will become slower over larger collections. This could be reduced to `O(log(n) + n)` if I implemented a binary search instead of simply iterating over every element, however depends on the key values being sort-able.

If speed of deletion is considered more important than preservation of insertion order, then I could omit the copying stage of the delete function, and make the insert method check for the first available location storing no value, or a `nullptr`.

C++ Support for Abstract Data Types